**Автоматное программирование**

Парадигма программирования, основанная на применении конечных автоматов для описания поведения программ.

Модель, с помощью которой удобно представлять процесс, имеющий конечное число дискретных управляющих состояний.

Одна из центральных идей автоматного программирования состоит в отделении описания логики поведения (при каких условиях необходимо выполнить те или иные действия) от описания его семантики (собственно смысла каждого из действий). Кроме того, описание логики при автоматном подходе жестко структурировано. Эти свойства делают автоматное описание сложного поведения наглядным и ясным.

**export** **default** (str) **=>** {

**let** result **=** '';

**let** state **=** 'outside'; *// outside, inside*

**for** (**let** i **=** 0; i **<** str.length; i **+=** 1) {

**const** symbol **=** str[i];

**switch** (state) {

**case** 'inside':

**if** (symbol **===** ' ') {

state **=** 'outside';

}

result **+=** symbol;

**break**;

**case** 'outside':

**if** (symbol **!==** ' ') {

state **=** 'inside';

result **+=** symbol.toUpperCase();

} **else** {

result **+=** symbol;

}

**break**;

}

}

**return** result;

};

И последнее. Что значит "управляющие состояния"? Понятие состояния не является чужеродным для мира программирования. В одной из первых лекций я рассказывал о том, что состояние программы это, грубо говоря, слепок её памяти. Другими словами, значение всех переменных в конкретный момент времени. Это действительно так, но можно пойти ещё дальше и заметить, что состояние можно поделить на два типа. Первый тип — это состояние, отвечающее за все возможные пути движения данных сквозь программу. Второй — это данные сами по себе или так называемое вычислительное состояние.

export default (text) => {

  const result = [];

  // before, inside, after

  let state = 'before';

  let word = [];

  Array.from(text).forEach((symbol) => {

    switch (state) {

      case 'before':

        if (symbol !== ' ' && symbol !== '\n') {

          state = 'inside';

          word.push(symbol);

        }

        break;

      case 'inside':

        if (symbol === ' ' || symbol === '\n') {

          result.push(word.join(''));

          word = [];

          state = symbol === ' ' ? 'after' : 'before';

        } else {

          word.push(symbol);

        }

        break;

      case 'after':

        if (symbol === '\n') {

          state = 'before';

        }

        break;

      default:

        throw new Error(`Unexpected state '${state}'`);

    }

  });

  if (word.length > 0) {

    result.push(word.join(''));

  }

  return result;

};